SAFELY NAVIGATING THE RBO-TO-CBO MINEFIELD

John Kanagaraj, DBSoft Inc

INTRODUCTION

Ever stepped on a land mine while getting from Point A to Point B in a remote battlefield? (Obviously, you haven't or you wouldn't be reading this, but we did get your attention, right?) At times, upgrades of Applications and Database versions is like navigating a minefield strewn with buried performance 'mines' that explode in the face of the unaware DBA. Many if not most of these 'mines' deal with the way Oracle chooses to access the data, which is the job of the Optimizer (but more of that later...)

In this paper, we will point out some of these 'mines' and provide you with some tips and techniques for uncovering them and rendering them ineffective. This also involves exploding some myths – most of them based on some commonly held misconceptions about the Oracle optimizers. This newfound revelation of the Optimizer will help you understand the tips and techniques that we will later suggest. We will *not* however deal with the actual task of migration of an application from RBO land to CBO heaven, nor will we deal with SQL tuning and hinting in depth. These subjects are deep enough to merit several discourses of their own and are thus outside the scope of this paper.

As a DBA, you will not (and should not) be expected to deal with the task of migration. You will however be called in to figure out why a particular program or task is now running 'horribly slowly' after such a migration, and this paper should help point out factors that could be the likely cause. Since these factors are under the realm of the DBA rather than the Developer, it is your responsibility to make sure that these are taken care of right from the beginning. Even if performance issues are not present, you are still responsible as a DBA for making sure that the environment related to the Optimizer is correctly setup and maintained. Understanding the likely problem areas of the optimizer as laid out in this paper will help making this possible.

THE BATTLEGROUND

The Oracle kernel relies on the Optimizer layer to determine the 'best access path' to the data – indeed the Optimizer is the heart of the SQL processing engine. Up until Version 7 of the Oracle database, there was only one access method – one based on a set of rigid *Rules*. With the introduction of Oracle 7.0 way back in 1990, Oracle introduced an additional access method – one based on the *Cost* of access. And thus was born the names that are both familiar yet bring out a twinge of apprehension - the older Rule based Optimizer (or the RBO as it was now known) and the new kid on the block, the Cost based Optimizer (or the CBO).

Although many customers started adopting the CBO in Version 7.3, for reasons that we discuss below the uptake was slow. So much so that even Oracle Corporation moved its large ERP suite – succinctly named 'Oracle Applications' – from RBO to CBO as recently as less than three years back. Many large applications (and third party ones in particular) are still using the safe-and-steady RBO, unwilling to risk taking on performance issues that may be brought by this reasonably complex change.

While this was going on, Oracle Corporation stopped improvising on the RBO as of Version 8.0 and indicated that all new performance improvements and features will work only with the CBO. At the same time, Database sizes, complex inter-networked and interconnected applications grew in parallel. While the newer database versions and CBO supported these large and complex databases, the RBO did not. The User – in the form of the DBA and the Developer – was now stuck with the task of migrating applications from the RBO to the CBO.

And there, between the RBO on one end and the CBO on the other, lies this virtual minefield. Your task as a DBA, if you choose to take it, is to safely *navigate* your applications through this minefield! We hope that this paper will serve as one that will help you in this task.

SOME ESSENTIAL DETAILS BEFORE WE START…

As indicated before, the Optimizer decides how data should be accessed when presented with a particular DML (SELECT, INSERT, UPDATE, DELETE or internal recursive SQL) statement. There are usually many different ways of executing the given SQL statement – for example using one or more indexes, choosing the type of join for multiple tables, etc. The final objective is to execute the statement most efficiently and using the least resources, given the circumstances and environment.

The RBO makes this process simple – a set of 15 rules is applied rigidly in an IF-THEN-ELSE logic tree. The access path is not altered based on the spread or size of the data, nor does presence of other optimizer parameters influence the RBO. The CBO on the other hand, considers its path based on the estimated Cost of access for that path. This cost is calculated using statistics about the objects, which in turn feeds some algorithms used for calculating the most efficient access path. The cost of access is calculated for multiple alternate paths, and the path that produces the least cost for the given goal is chosen. This cost is thus dependent on

- The statistical values for the objects involved (for example sizes and count of Columns, Tables, Indexes, Histograms)
- The algorithms used to transform these numerical statistics into 'Cost of access'

While both the RBO and CBO are available in the RDBMS, the choice of which one is used to decide the access path is based on a number of parameters:

- The choice is decided by certain settings at the various levels, with the setting at the lower levels overriding the settings at the higher levels:
  - At the Instance level, which also is the highest level, the choice of which Optimizer to use is decided by the 'OPTIMIZER_MODE' initialization variable. A choice of RULE (RBO only where applicable), CHOOSE (CBO when statistics are available, RBO otherwise), ALL_ROWS (best throughput), FIRST_ROWS (best response) and FIRST_ROWS_n (new in 9i) are available.
  - At the Session level, this is decided by the OPTIMIZER_GOAL parameter of the ALTER SESSION statement. Again, all the above choices can be provided.
  - At the SQL level – the lowest– this is decided by Hints buried in the SQL itself
- The choice is also defaulted by the presence (or absence) of statistics when the OPTIMIZER_MODE/GOAL is set to CHOOSE.
- Regardless of the mode or presence of statistics, the type of object being accessed also determines the mode when using Oracle 8 and upward

The object statistics used by the CBO is generated by both the ANALYZE command as well as the DBMS_STATS internal package (8i and up). While the ANALYZE command would work on any of the current as well as older versions, the use of DBMS_STATS package is highly recommended. Oracle Applications provides a FND_STATS package that internally uses DBMS_STATS but stores some additional information elsewhere. Regardless of which one is used, all of them perform the same basic function – that of collecting the following object statistics:

- Table statistics (includes Partitioned tables and IOTs)
  - Number of rows (by partitions where applicable)

  - Number of blocks
  - Number of empty blocks
  - Average row length
  - Chained row count (ANALYZE only!)
- Column statistics
  - Number of distinct values (NDV) in column
  - Number of nulls in column
  - Data distribution (histogram)
- Index statistics
  - Number of leaf blocks
  - Levels
  - Clustering factor

Using both methods, these statistics can be computed exactly using the COMPUTE clause or estimated using the ESTIMATE clause. More on this later…

With the exception of the Histogram statistics, these statistics are stored along with the object descriptive internal tables and exposed via the 'ALL_', 'DBA_' and 'USER_' set of views. The LAST_ANALYZED and SAMPLE_SIZE columns present in most of these views exposes the last date/time when statistics were gathered and the sample size used at that time.

Ostensibly, the CBO is the best choice since it adjusts itself to choose the best path depending on the data in addition to being able to change it's path based on external 'inputs'. In reality though, there were (and still are) some issues using or migrating to the CBO.

SO WHAT *WAS* THE PROBLEM WITH THE CBO?

As with a coin, there are two sides to this story. The core of the CBO consists of the object statistics and the cost calculations and algorithms that operate on it. We will map them to the two entities involved - Oracle on the one hand and the User on the other:

- Incorrect or incomplete statistics collection on the User side
- Bugs, changes, 'instability' of algorithms between versions on the Oracle RDBMS side

Lack of detailed documentation and understanding on both sides, the cost and effort involved in changing applications and just plain caution contributed to slow uptake of the CBO. Myths around these half-facts then grew up and took deep root…

Consequently, many RBO-to-CBO migrations were delayed and some were even abandoned. In many cases, after such a migration, upgrades of Database versions resulted in cost algorithm changes, adversely affecting performance. As we indicated before, even Oracle delayed this migration in it's own complex ERP package until recently. It is widely held in performance-related DBA circles that the CBO grew up in Version 8.0 and has really matured in 8i, with more to come in 9i. If you had previously postponed or abandoned such a migration but are now at Oracle 8i or above, we would strongly recommend that you reconsider the migration. We hope this paper will help you to understand what went wrong before and how to overcome what you might face.

THE BURIED 'MINES'

In this section, we will list many of the myths about the Optimizers and our reasoning that exposes it. Where possible, we will also include true examples and data to support these claims. We want to make you aware of these myths as you may face some of them when you navigate your own 'minefield'. Some of these myths were truths once upon a time, but are no longer relevant or true now. Hence, they may ring true depending on where you are in terms of versions and configurations.

MYTH: "RBO WILL ALWAYS BE FORCED BY HINT OR MODE"

This is one of the top myths, and usually manifests itself during the process of migration. The RBO will not be forced *always*, even if the OPTIMIZER_MODE or RULE Hint is set. When any of the following objects or operations are present, the CBO is forced regardless:

- Partitioned tables and indexes
- Index-organized tables
- Reverse key indexes
- Function-based indexes
- `SAMPLE` clauses in a `SELECT` statement
- Parallel execution and parallel DML (i.e. presence of DEGREE on Tables/Indexes)
- Star transformations
- Star joins
- Extensible optimizer
- Query rewrite (materialized views)
- Progress meter
- Hash joins
- Bitmap indexes and bitmap join indexes
- Partition Views
- Index skip scans

The solution is to carefully consider access methods, especially when starting to use new features such as the ones above.

MYTH: "THE CHOICE OF THE OPTIMIZER IS DECIDED BY ABSENCE OF STATISTICS"

This is another common myth, i.e. when the OPTIMIZER_MODE is set to CHOOSE and Statistics are not present, it is assumed that the RBO is used. Let us offer a small but important addition to this statement: Presence of objects statistics on *any* of the involved objects will trigger the CBO rather than RBO. When the CBO detects this situation, it calculates an estimated value for the missing statistics based on some internal assumptions shown in Table 1 below:

Statistic	*Default Value Used by Optimizer*
Tables	
Cardinality	100 rows

Avg. row length	20 bytes
No. of blocks	100
Remote cardinality	2000 rows
Remote average row length	100 bytes
Indexes	
Levels	1
Leaf blocks	25
Leaf blocks/key	1
Data blocks/key	1
Distinct keys	100
Clustering factor	800 (8*no. of blocks)

Table 1

These 'guesstimates' are way off the mark in most cases. For example, the cardinality (number of rows in a row set) is assumed to be 100 when this could have been say 10000 (whoever heard of a 100 row transaction table!). This results in making a Full table scan (FTS) look less costly (since the figures are small) and thus what could have been accessed via an Index scan would blow up into an FTS at the wrong time… You will also observe that assumptions are made about *remote* databases – this implies that you will need to keep in mind that presence (or absence) of statistics matter even on databases connected via Database Links.

Such considerations are especially important when migrating portions of an interconnected database, even if done one manageable portion at a time. Ideally, you need to make sure that all related portions migrate to the CBO world at one point of time, including remote databases involved in the equation. (Not as easy as you thought it was, right?)

MYTH: "ANALYZE THE *ENTIRE* DATABASE, INCLUDING SYS"

This is usually the result of an over-enthusiastic move to the CBO. The internal Data dictionary (mostly owned by SYS) is heavily optimized for the RBO, a carry over from the days of Oracle 6 when RBO was the only child in the family. Many Data dictionary views are hinted by RULE, but some are not. Messing around with them is not good for the health of the Database! On a more serious note, Database deadlocks have been known to occur when analyzing the SYS schema as rows being inserted into the Histogram-related internal tables lock themselves out. For further details, refer to Metalink Note 35272.1.

An interesting side note to this myth is the hidden issue with the DBMS_UTILITY.ANALYZE_DATABASE procedure. Invocation of this in-built package/procedure used to generate statistics for *all* users, including SYS. Bug 969814, released as late as 8.1.7, fixes ANALYZE_DATABASE so it does not analyze the dictionary tables FET$ and UET$.

MYTH: "COMPUTE IS BETTER THAN ESTIMATE"

This one generates an endless debate actually, so we will not take a firm stand either way. Rather, we will present some figures that throw some light on the issue and allow us to step back and look at the situation. The problem with COMPUTE is that it has to scan the entire table, sort it and figure out the exact data distribution. On the other hand, ESTIMATE steps through samples of the data, sorts and analyzes only a portion of the data.

In a recent test for the effectiveness of COMPUTE versus ESTIMATE on a *static* clone of a reasonably large Oracle Apps database, the statistics were generated and stored for both COMPUTE and ESTIMATE. The Database consisted of about

3,300 tables and 6,000 indexes and occupied approximately 120 Gb. The ESTIMATE percentage was defaulted to 10% and no activity other than ANALYZE was allowed on this clone during the entire period. Table statistics including row count, average row length and blocks occupied were analyzed. This showed that there were *some* differences in row count and average row length on 321 of these tables. Row count differences ranged from a value of 53 row less in the ESTIMATE of a table containing 205,743 rows (0.025%) all the way up-to a count difference of 101,704 in 13,311,090 rows (0.76%). Even assuming a difference of a maximum of 5% in these scenarios, you are not far off the goal. Further analysis showed that a smaller average row length coupled with a small table produced larger variations than was usually seen.

The differences however, were far more pronounced in Indexes – differences of upto 300% were noticed. Further analysis showed that this was related to the percentage of deleted leaf rows in the index. If this percentage is high, the possibility of ESTIMATE going wrong was also high, as the deletions are not factored in correctly. This was especially true if the deletions occurred in leaf blocks that were probably not involved in the ESTIMATE. When the deleted leaf rows was low or even nil within the index, the percentage difference was much lower, in the range of 4 to 5%.

The real myth killer is the *cost* of COMPUTE versus ESTIMATE - COMPUTE required 66,553,308 reads versus 38,951,158 reads for ESTIMATE – almost 70% more reads for COMPUTE. The sorting involved in determining the averages and data distribution was a clincher – COMPUTE processed 4,263,724,259 rows in sorting operations while ESTIMATE sorted just 18,025,069 – i.e. about 235% more rows were sorted for the COMPUTE operation. The last nail in the coffin was the time taken to COMPUTE statistics - about 36 hours against the time to ESTIMATE of just 12 hours.

While the figures speak for themselves, we will offer some general advice to the cautious: ESTIMATE on tables and COMPUTE on Indexes. Columns are analyzed by default, but serve no useful purpose other than showing data spread. Hence, you could ANALYZE only Tables and Indexed columns alone. An identified list of 'small' tables could also be COMPUTED rather than ANALYZED. This advice is given because ESTIMATE on a table comes close as far as row count goes, while COMPUTE on Indexes generates a more accurate picture of both data distribution as well as object size statistics. Testing the effectiveness of COMPUTE versus ANALYZE is simple and provides you with figures that you can use to decide the strategy for your situation.

Before we move to the next topic, keep in mind that an ANALYZE/ESTIMATE with a sample size greater than or equal to 50% will result in COMPUTE.

MYTH: "ANALYZE IS BETTER THAN USING DBMS_STATS"

This myth probably grew on its own since DBAs felt comfortable using something they knew and have already worked with. Scripts that did the job were already available – why break something that is working well? The answer lies in handling some of the new object structures such as Partitions and IOTs that can be done only using DBMS_STATS. In addition, DBMS_STATS enables collection of Statistics in parallel on Partitioned tables which are usually huge – ANALYZE does not parallelize its operations. DBMS_STATS can be used to manipulate statistics – Statistics can be exported/imported, and values can be set and saved as well. DBMS_STATS can also collect object statistics for 'stale' objects. It can also adjust itself to collect better statistics for the CBO depending on algorithmic changes. Further, Oracle 9i supports the use of the AUTO_SAMPLE_SIZE parameter in DBMS_STATS which promises statistical accuracy while maximizing performance. The only value that DBMS_STATS does not collect when compared to ANALYZE is row chaining information – so use ANALYZE specifically for tables that historically experience chaining.

While you may choose not to write off ANALYZE immediately, it would be worth the effort to move to DBMS_STATS based statistics collection. The FND_STATS package to collect Statistics in Oracle Applications 11i Databases is highly recommended – this function uses DBMS_STATS internally and stores certain statistics in FND_ tables.

MYTH: "GENERATE STATISTICS *EVERY* DAY FOR ALL OBJECTS"

This myth was probably created by an overly cautious attitude. While it is essential to have up-to date statistics, it is sometimes not necessary to re-create the statistics everyday. As with most situations, it all 'depends', mostly on the volatility of data. In this respect, Oracle 8i provides a facility to monitor designated tables for DML activity that changes data. When an 'ALTER <table> MONITORING' is issued on a table (or a set of tables), the kernel tracks the approximate number of inserts, updates and deletes to that table in a special structure in the SGA. This information is flushed to the data dictionary by SMON periodically (once in 3 hours) and at normal database close, and is exposed via the DBA_TAB_MODIFICATIONS view. Once this is obtained over a period of time, you can then decide to either setup your own list of ANALYZE statements for a refresh period of *your well-informed* choice. Alternately, this can also be gathered via the 'GATHER STALE' parameter in DBMS_STATS.

Setting this up kind of monitoring imposes minimal overheads since this is memory based, and the gains obtained by *not* over analyzing the database more than makes up for this overhead.

MYTH: "HISTOGRAMS ARE USELESS"

This myth applies to the more advanced user who has already figured out that Histograms are used only when SQL containing hard coded values is parsed. Although this is true, Histograms do capture data 'skew' in columns of interest. Storage and analysis of the Histograms can potentially reveal many interesting trends for Data Analysts. On the surface, this seems to imply that users should *not* use bind variables (horrors!) and fill the Shared pool with 'parsed-once-never-used-again' SQL snippets. However, Oracle has kindly addressed this issue in Oracle 9i – to quote from the Fine Manual:

"The CBO now peeks at the values of user-defined bind variables on the first invocation of a cursor. This lets the optimizer determine the selectivity of any `WHERE` clause condition, as well as if literals had been used instead of bind variables. When bind variables are used in a statement, it is assumed that cursor sharing is intended and that different invocations are supposed to use the same execution plan."

This gives us the best of both worlds! In other words, histograms are considered even when bind variables are used.

MYTH: "LET INIT.ORA PARAMETERS DEFAULT"

Oracle initialization parameters default to some internally defined values when they are not specified in the 'init<SID>.ora' file. A growing number of these parameters drastically modify the behavior of the CBO. Among these, the OPTIMIZER_INDEX_CACHING and OPTIMIZER_INDEX_COST_ADJ affect the use (or non-use!) of Indexes the most. Default values influence the CBO to use Full Table Scans (FTS) rather than Indexed reads. (This has resulted in some DBAs calling the CBO by many names, the kindest of which was 'FTS friendly'!)

The OPTIMIZER_INDEX_CACHING parameter indicates the percentage of the index blocks the optimizer should assume is available in the cache. Setting this parameter to a higher value makes nested loops joins and IN-list iterators look less expensive to the optimizer. Therefore, it will be more likely to pick nested loops joins over hash or sort-merge joins, and to pick indexes using IN-list iterators over other indexes or full table scans. This is left to default to 0, meaning that the possibility of finding cached Index blocks is nil! As a starting point, this could be safely set to a high value such as 80, if not 50.

The OPTIMIZER_INDEX_COST_ADJ parameter lets you tune optimizer behavior for access path selection to be more index friendly. In other words, it makes the optimizer more (or less) prone to selecting an index access path over a full table scan. The default for this parameter is 100 percent, at which the optimizer evaluates index access paths at the regular cost. Any other value makes the optimizer evaluate the access path at that percentage of the regular cost. For example, a setting of 50 makes the index access path look half as expensive as normal. A starting value of 50 is usually a good bet.

There are many other parameters - visible and hidden - that influence the CBO, but we will not discuss them here for the sake of brevity. For these parameters, it may still be a good idea to let them default, unless testing proves otherwise. A list of these currently applicable parameters can be easily determined by setting a 10053 event and analyzing the trace file generated consequently. Such a list from the trace file in an 8.1.6 database is shown below, only from the point of view of exposing the name and number of such parameters:

```
***************************************
PARAMETERS USED BY THE OPTIMIZER
********************************
OPTIMIZER_FEATURES_ENABLE = 8.1.6
OPTIMIZER_MODE/GOAL = Choose
OPTIMIZER_PERCENT_PARALLEL = 0
HASH_AREA_SIZE = 4194304
HASH_JOIN_ENABLED = TRUE
HASH_MULTIBLOCK_IO_COUNT = 0
OPTIMIZER_SEARCH_LIMIT = 5
PARTITION_VIEW_ENABLED = FALSE
_ALWAYS_STAR_TRANSFORMATION = FALSE
_B_TREE_BITMAP_PLANS = FALSE
STAR_TRANSFORMATION_ENABLED = FALSE
_COMPLEX_VIEW_MERGING = FALSE
_PUSH_JOIN_PREDICATE = FALSE
PARALLEL_BROADCAST_ENABLED = FALSE
OPTIMIZER_MAX_PERMUTATIONS = 80000
OPTIMIZER_INDEX_CACHING = 0
OPTIMIZER_INDEX_COST_ADJ = 100
QUERY_REWRITE_ENABLED = FALSE
QUERY_REWRITE_INTEGRITY = ENFORCED
_INDEX_JOIN_ENABLED = FALSE
_SORT_ELIMINATION_COST_RATIO = 0
_OR_EXPAND_NVL_PREDICATE = FALSE
_OPTIMIZER_MODE_FORCE = TRUE
_OPTIMIZER_UNDO_CHANGES = FALSE
_UNNEST_SUBQUERY = FALSE
_PUSH_JOIN_UNION_VIEW = FALSE
_FAST_FULL_SCAN_ENABLED = TRUE
_OPTIM_ENHANCE_NNULL_DETECTION = TRUE
_ORDERED_NESTED_LOOP = FALSE
_NESTED_LOOP_FUDGE = 100
_NO_OR_EXPANSION = FALSE
_QUERY_COST_REWRITE = TRUE
QUERY_REWRITE_EXPRESSION = TRUE
_IMPROVED_ROW_LENGTH_ENABLED = TRUE
_USE_NOSEGMENT_INDEXES = FALSE
_ENABLE_TYPE_DEP_SELECTIVITY = TRUE
_IMPROVED_OUTERJOIN_CARD = TRUE
_OPTIMIZER_ADJUST_FOR_NULLS = TRUE
_OPTIMIZER_CHOOSE_PERMUTATION = 0
_USE_COLUMN_STATS_FOR_FUNCTION = FALSE
_SUBQUERY_PRUNING_ENABLED = TRUE
_SUBQUERY_PRUNING_REDUCTION_FACTOR = 50
_SUBQUERY_PRUNING_COST_FACTOR = 20
DB_FILE_MULTIBLOCK_READ_COUNT = 16
SORT_AREA_SIZE = 4194304
```

MYTH: "HIGH VALUES FOR MULTI_BLOCK_READ_COUNT AND SORT_AREA SIZES IS GOOD"

This is related to the previous myth in some ways and it goes thus: "High values for parameters such as DB_FILE_MULTIBLOCK_READ_COUNT, SORT_AREA_SIZE and SORT_AREA_RETAINED_SIZE is a *good* thing, because we have lots of memory and a good I/O system in the form of RAID/SAN cache". This is partly true – you do need to size these values correctly for a well-tuned Database. However, large values for these parameters *may* influence the optimizer to favor Full Table Scans (FTS) and sorts over other more efficient methods. It does this by making FTS look less expensive than they actually are. On the other hand, you may not be using the I/O and memory available to you if you set too

low a value. As explained above, if you set too high a value you may be influencing the optimizer incorrectly. The trick is to arrive at the 'best' value for your situation.

In the case of the DB_FILE_MULTIBLOCK_READ_COUNT, set it so that the product of this parameter and the DB_BLOCK_SIZE matches the lowest of the following: The Operating System limit for a single read, the Volume Manager maximum or the hardware cache that you may have. In most cases, this defaults to 64K or 128K. Refer to Metalink Note:1037322.6 for further details. In the case of the other parameters, make sure that you lean neither way, but understand what that parameter would influence and be aware that it may also influence the optimizer.

MYTH: "SETTING UP UNDOCUMENTED PARAMETERS IN INIT.ORA IS BAAAD!"

Undocumented initialization parameters are those that do not appear in the Oracle manuals (which effectively renders them 'undocumented'!). They are generally characterized by having a name that starts with an underscore, which makes DBAs refer to them as 'underscore' parameters. Sometimes they are also called 'hidden' parameters. Some of these parameters are harmless – for example the '_trace_files_public = TRUE' makes the DBAs life easier by assigning read permission on Oracle trace files to all users. However, several of them influence the optimizers and have the potential to strew some mini-mines on their own. Some of these hidden parameters were listed in a previous section.

While you should *never* setup undocumented parameters without proper authorization or recommendations from Oracle support, it is possible that they are a requirement for certain applications. In particular, Oracle Applications 11i (11.5.x) requires the use of the following hidden parameters:

```
_optimizer_undo_changes         = false
_optimizer_mode_force           = true
_complex_view_merging  = TRUE
_push_join_predicate   = TRUE
_sort_elimination_cost_ratio    = 5
_use_column_stats_for_function  = TRUE
_like_with_bind_as_equality     = TRUE
_or_expand_nvl_predicate       = TRUE
_push_join_union_view           = TRUE
_table_scan_cost_plus_one       = TRUE
_fast_full_scan_enabled         = FALSE
_ordered_nested_loop            = TRUE
_sqlexec_progression_cost=0
```

For those that are interested, the only explanation for these parameters are found in the 'ifilecbo.ora' file that is sourced from $AD_TOP/admin/sql. The contents may vary between versions of the underlying Oracle version as well as the Apps version. This file is referred from the main init.ora file via an 'ifile' include directive and is thus subject to 'transparent' changes.

These 'underscore' parameters are also characterized by sudden appearances, disappearances (and re-appearances in some cases!) in different versions – and therein lies the main problem. They 'appear' when it is deemed necessary to correct or influence some part of the optimizer code in one version, and 'disappear' in the next version when that particular failing is corrected. (Obviously, they 're-appear' when the fix doesn't work as it was expected to!) The point we want to make here is that while these hidden parameters may need to be set as required by Oracle support, they do need to be reviewed during a subsequent upgrade. Thus, they are not necessarily as evil as they are made out to be but have the potential to become either your best friend or your worst enemy!

MYTH: "HINT ALL YOU CAN!"

Hints in the SQL can be used to 'force' the CBO to adopt a certain path. While it is a good thing that Hints do override the seemingly incorrect path chosen, they cannot - and should not - be a *final* solution. You should remember that Hints are workarounds, and they may not be required in the next version when a bug or 'feature' in the current release is fixed. Hints should be reviewed and removed if necessary during version or patch upgrades, as they *may* hold back the Hinted SQL from performing at its best. Hints don't adapt to the changing environment – either in the data or in the operating environment. The reasoning behind the hints are sometimes 'forgotten' and thus, they become permanent. Our advice is to use Hints as a last resort and review them at times when the changes mentioned above occur. Also, keep in mind that Hints cannot be tacked onto third party, canned products.

CLEARING THE MINEFIELD

Now that you know what the mines are and where they are buried, let us suggest some guidelines for removing them safely. These techniques are mostly pro-active steps that need to be taken before you step into this 'minefield'. However, if you do find yourself in the situation of having to jump into one, knowing a few tools and techniques can help. In no particular order, they are:

FEEDING THE BEAST – SERVE UP THE STATISTICS

As we mentioned before, many of the 'problems' with the CBO manifest themselves because of stale or missing object statistics. It is essential that the object statistics are kept up-to date, and in a timely, periodic fashion. Following the suggestions listed above to keep your statistics up to date will result in a well-tuned and balanced database. It is essential that *all* objects (with the exception of SYS objects – the Data Dictionary!) in the database have statistics – even if you feel that they are not involved.

A PICTURE FROM THE PAST – PRESERVE HISTORICAL STATISTICS

While collecting and maintaining object statistics is vital, it is also advisable to store these statistics to enable trend analysis in the future. As indicated before, DBMS_STATS enables you to store these statistics using the IMPORT_<level>_STATS parameter. Alternately, you can use your own Statistics tables that capture the essential details from the required tables such as DBA_TABLES, DBA_INDEXES, DBA_HISTOGRAMS, etc. The latter provides a good insight into column level changes over time.

While object statistics are one part of the picture, it is essential to collect, store and analyze Database and Operating System statistics as well. Most UNIX systems offer a combination of sar, vmstat and iostat commands that can be used to collect OS statistics. NT offers the Perfmon utility to collect OS statistics. No DBA Toolkit should be complete without the STATSPACK in-built Database statistics collection utility. Further details about the STATSPACK utility are available in the Oracle Magazine archives. Suffice to say that the reporting period should be granular enough to be able to narrow down problems in a given time-period.

Scripts built around these utilities can glean useful statistics. When stored and compared with past and current values, the reasons for a number of problems can surface. As long as new applications have not been added or existing ones changed drastically, you should expect to see normal and incremental changes. When an upgrade goes wrong, you will usually notice a drastic change in some of these measurements. Such changes should be analyzed and the reasons understood.

In addition to these pure numbers, it is also essential to collect baseline performance for important processes from users. You should have established current response times for critical online transactions as well as time-to-complete for essential batch

processing jobs. This will enable comparison after upgrades or changes and ensure that you have a numerical baseline to work from or aim for.

CUT YOUR TEETH IN BOOT CAMP – A TEST DATABASE

It is essential that you create and maintain a Test environment. This is normally overlooked in the scheme of things, and at many sites is an after thought. Even if it were implemented, financial constraints don't normally allow the Test environment to match the Production environment. Although it would be ideal to maintain a test environment that is an exact copy of the production environment, it need not be. However, it is essential that the following be done without fail:

- The test environment should be of the same platform, Operating System and Database version as the Production environment and have at least two CPUS and the same amount of disk space. Don't scrunch up on disk space – it is cheap when placed outside a SAN or special RAID array. Any such low-cost environment will do, as long as adequate amounts of disk space is available.
- This database and application instance should be dedicated to performance testing only. Mixing this with a 'Development/User acceptance' requirement will result in incorrect measurements as code keep changing
- The test database should be cloned periodically from Production, preferably using a Hot/Cold backup so that the same file/object spread is maintained
- Init.ora and other OS parameters should match Production, unless changes are being validated
- Response time and other measurements can then be *extrapolated* – the end result in Production is expected to be better than the one in Test. We will look at some internal 'V$ ' views that will expose what's going on within the database in this regard
- The test instance should have the same mechanism that tracks and controls change in your organization (and this leads us to the next point!)

WHAT WAS CHANGED AND WHEN? – TRACKING YOUR PATH

It is essential to have some form of Change control and tracking in your environment. This allows you to document, schedule, implement, monitor and rollback changes if required. It is also the living history of your environment. Overlaying your statistical figures by this historical map will usually show up any changes that resulted in problems. We cannot over emphasize this essential but often overlooked IT process.

TAKING IT ON - PIECE BY PIECE

As far as possible, you should implement changes one piece at a time. This way, you control the amount of change and the inevitable set of problems that come along with it. Sometimes, it is not possible to perform such changes in a piece-meal fashion, and that is when implementing it in a test environment to work out solutions beforehand becomes an absolute essential.

THE HERE-AND-NOW – TOOLS TO MONITOR AND DETECT PROBLEMS

In addition to collecting statistics from Database and Operating System tools, you should implement ongoing monitoring tools. Both Oracle supplied tools such as OEM and third-party tools monitor all aspects of the Database and Operating system. Even if you don't have these tools, it is quite easy to build up a personal library of scripts that will do the job of monitoring. If you do detect a problem, the following internal views expose the details of what's going on within the database:

- V$SYSTEM_EVENT – Collective waits by event since startup. When comparing this between Production and Test, for a given load, the average figures may vary. However, the number and type of waits may correspond with one another and this is key to the analysis of the problem

- V$SYSSTAT – Collective database statistics since startup. In a comparison between Production and Test databases, most of these figures should match for a given load. For example, given the same data and parameters, the number of rows fetched via table scan and rowid should be similar, even if the process took longer to execute in Test as compared to Production.
- V$SESSION_WAIT – Session-wise wait. This exposes the current wait events for a given session. Rolling this up and summarizing by event is an excellent way of determining the current performance problem in an instance
- V$SESSION_EVENT – Session-wise database statistics since session startup. Exposes the amount of work done by that session
- VSQL, VLATCH, V$FILESTAT and related DBA_ views can then be used to drill down further.

You should get know these views and be able to use them effectively. They provide invaluable information about the 'here-and-now' situation. Further details about these events and their meaning can be obtained from Appendices A through C in the Oracle 8i Reference Manual. STATSPACK uses these views to produce detailed reports. Note that setting 'TIMED_STATISTICS=TRUE' is necessary to be able to obtain timings for some of these statistics.

CATCH 'EM WHEN THEY COME IN – USING THE ON-LOGON SYSTEM TRIGGER

Oracle 8i (and above) provides a System level trigger that is invoked when a user logs on. You can easily setup a system level LOGON trigger that can change the environment for a particular user or situation. This is enabled via execution of the ALTER SESSION statement with the SET <parameter> = <parameter_value> clause, the result of which is applicable only for the lifetime of that session. A limited set of parameters can be changed at session level via this command. (On a 8.1.6 database 72 out of a total of 215 parameters could have been changed at the session level. It is interesting to note that this list includes a number of hidden parameters as well!) This technique can then be used for either experimenting with parameter changes on a set of users or schemas without affecting the entire database. On the other hand, it could also be used to 'fix' the optimizer environment for a specified user or schema while the rest of the Database chugs along with existing defaults. When implemented properly, it even helps in performing controlled tests in Production.

WHEN ALL ELSE FAILS – HINTS AND PLAN STABILITY

When your best efforts fail to solve the problem, you can then try to use a combination of 'Plan Stability' and Hints within SQL that is misbehaving. Hints to the Optimizer are 'hardcoded' within an SQL statement – this forces the SQL to adopt an access path that you have specified. There are many hints, and all of them are clearly described in both the manuals and specialized tuning books.

When faced with a third-party application that cannot be hinted, use Plan Stability. Essentially, Plan stability directs the optimizer to check some internal views for SQL that has previously been stored and follow a fixed path when parsing this SQL. The details of how this is implemented is outside the scope of this paper, but you can refer the 'Oracle 8i Designing and Tuning for Performance Manual' as well as the Concepts and Admin manuals for details. Oracle 9i allows you to edit these outlines, and that is something new to consider as you go forward.

LOOKING FORWARD – 9I IS HERE (ALMOST!)

The newly announced Oracle 9i database provides some exciting new features in the area of performance management. You should plan to use these new features when you consider migrating to this new environment. Chief among these features is:

- FIRST_ROWS_n hint – this allows us to set the optimizer goal for best response time to fetch 'n' rows
- Values in bind variables considered – the optimizer now considers values in bind variables prior to parse

- DBMS_STATS collects and uses System statistics – CPU_COST and IO_COST are now collected and considered during access path determination
- New Optimizer hints (didn't we have enough!)
- Outline editing – you can now edit the outlines that are created as a template for plan stability
- ALTER INDEX MONITORING – Index usage (and non-usage) can now be tracked
- CURSOR_SHARING can now be set to SIMILAR for force sharing of similar statements by replacing literals by with system-generated bind variables

CONCLUSION

Migrating an application or a database from RBO to CBO is not trivial. A number of factors and myths need to be understood and tools and techniques described above have to be implemented before attempting such a migration. Even if such a migration was successfully done before, a subsequent application or database version upgrade can go awry depending on the use or non-use of new features in the Database. Moreover, even if you have a well-performing database, it is still a good idea to check it out against what we have revealed here.

Successful DBAs do not rest on their laurels but continuously strive to improve both their knowledge and the performance of the databases in their charge. It is our hope that we have aided you by shining a little light on what's going on 'under-the-covers' and helped dispel inaccuracies in understanding and implementation.

A footnote before we close – The use of Mines and Minefields, Battlefields and Explosions in this paper was not meant to evoke scenes of blood and gore, but purely to establish parallels. We are against war and all that it brings along with it (and leaves behind too in the form of broken families and mines that continue to explode long after the war is over).

ABOUT THE AUTHOR

John Kanagaraj is a Principal Consultant with DBSoft Inc., and resides in the Bay Area in sunny California. He has been working with various flavors of UNIX since '84 and with Oracle since '88, mostly as a Developer/DBA/Apps DBA and System Administrator. Prior to joining DBSoft, he led small teams of DBAs and UNIX/NT SysAdmins at Shell Petroleum companies in Brunei and Oman. He started his Troubleshooting career as a member (and later became head) of the Database SWAT/Benchmarking team at Wipro Infotech, India. His specialization is UNIX/Oracle Performance management, Backup/recovery and System Availability and he has put out many fires in these areas along the way since '84! John can be reached via email at 'ora_apps_dba_y@yahoo.com'.

REFERENCES

1. Numerous, spirited discussions in publicly accessible DBA Forums
2. Oracle 7.3/8i/9i manuals including the Reference Manual, SQL Reference manual and Tuning Guides
3. Metalink Notes (restricted to authorized users only)
4. Publicly accessible websites of Performance Gurus such as Steve Adams (http://www.ixora.com.au), Cary Millsap (http://www.hotsos.com), Jonathan Lewis (http://www.jlcomp.demon.co.uk) and Tim Gorman (http://www.evdbt.com) among others. Websites are listed in no particular order!